Sharded Counters

Suresh Podeti

7 min read · Oct 26, 2023

Introduction

Real-time applications like Facebook, Twitter, and YouTube have high user traffic. Users interact with the applications and perform multiple operations (**view, like, comment**, and so on) depending on the application's structure.

How will we handle millions of write requests coming against the likes on thousands of tweets per minute? The challenge is that writing takes more time than reading, and **concurrent** activity makes this problem harder. As the number of concurrent writes increases for some counter (which might be a variable residing in a node's memory), the **lock contention** increases non-linearly. After some point, we might spend most of the time acquiring the lock so that we could safely update the counter.

The above scenario shows how a simple counting operation becomes challenging to manage with precision and performance. This problem is known as the **heavy hitters problem.**

Distributed counter

What will happen when a single tweet on Twitter gets a million likes, and the application server receives a write request against each like to increment the relevant counter?

If we increment the counter concurrently, we would end up having inconsistencies in the data.

One of the possible solution: These millions of requests are eventually **serialized in a queue** for data consistency. Such **serialization** is one way to deal with concurrent

activity, though at the **expense of added delay**. Real-time applications want to keep the quality of experience high by providing as minimum as possible latency for the end user.

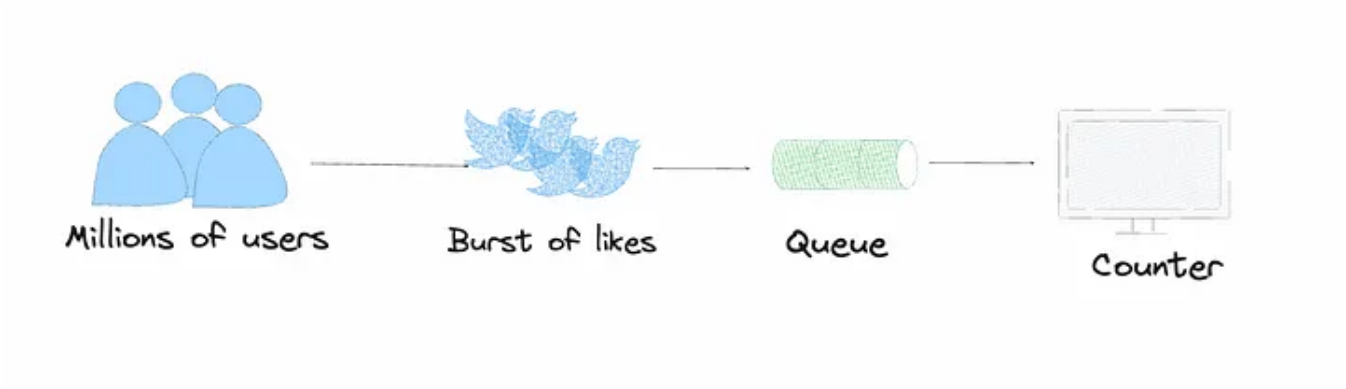

Fig 1.0: Burst of likes queued

A single counter for each tweet posted by a celebrity is not enough to handle millions of users. The solution to this problem is a **sharded counter**, also known as a **distributed counter**, where each counter has a specified number of shards as needed. These shards run on different computational units in parallel. We can improve performance and reduce contention by balancing the millions of write requests across shards.

First, a write request is forwarded to the specified **tweet counter** when the user likes that tweet. Then, the system chooses an available shard of the specified tweet counter to increment the like count. Let's look at the illustration below to understand sharded counters having specified shards:

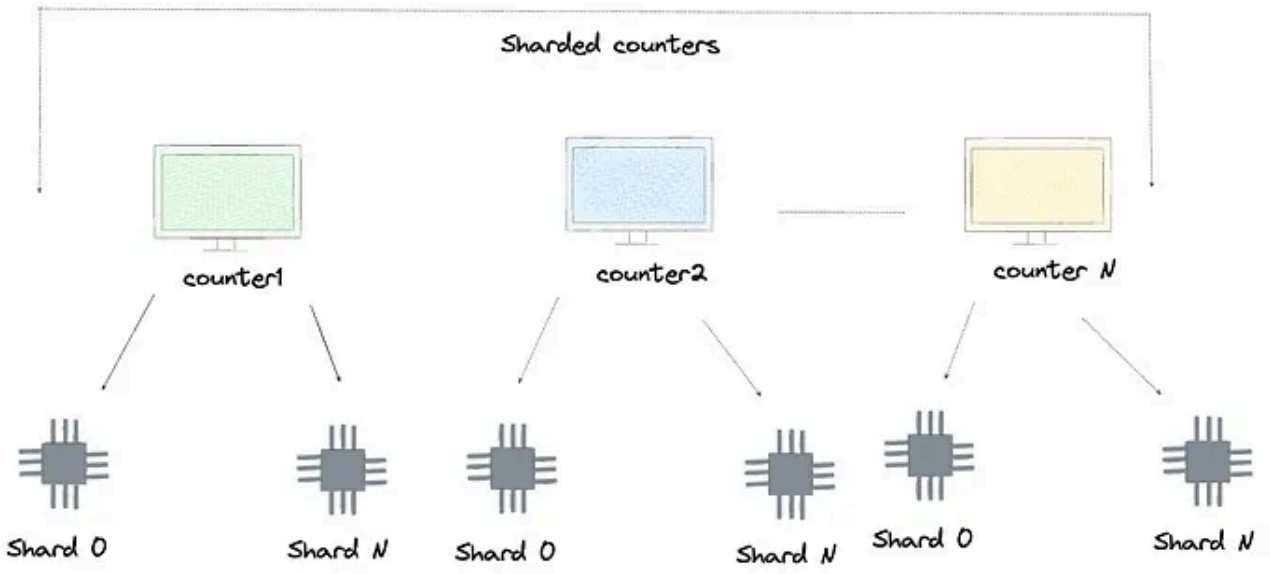

Fig 2.0: Counters and their shards working on different computational units

In the above illustration, the total number of shards per counter is (*N*+1). We'll use an appropriate value for *N* according to our needs.

Let's assume that a famous YouTube channel with millions of subscribers uploads a new video. The server receives a burst of write requests for video views from worldwide users. **First, the system creates multiple counters for the newly uploaded video**. The server forwards the request to the corresponding counter, and our system chooses the shard randomly and updates the shard value, which is initially zero. In contrast, when the server receives read requests, it adds the values of all the shards of a counter to get the current total.

We can use a sharded counter for every scenario where we need scalable counting (such as Facebook posts and YouTube videos).

Design

Sharded counter creation

Question is how does the system decide the number of shards in each counter?

The number of shards is critical for good performance. If the shard count is small for a specific write workload, we face high write contention, which results in slow writes.

On the other hand, if the shard count is too high for a particular write profile, we encounter a higher overhead on the read operation. The reason for slower reads is because of the collection of values from different shards that might reside on different nodes inside geographically distributed data centers. The reading cost of a counter value rises linearly with the number of shards because the values of all shards of a respective counter are added. The writes scale linearly as we add new shards due to increasing requests. Therefore, there is a trade-off between making writes quick versus read performance.

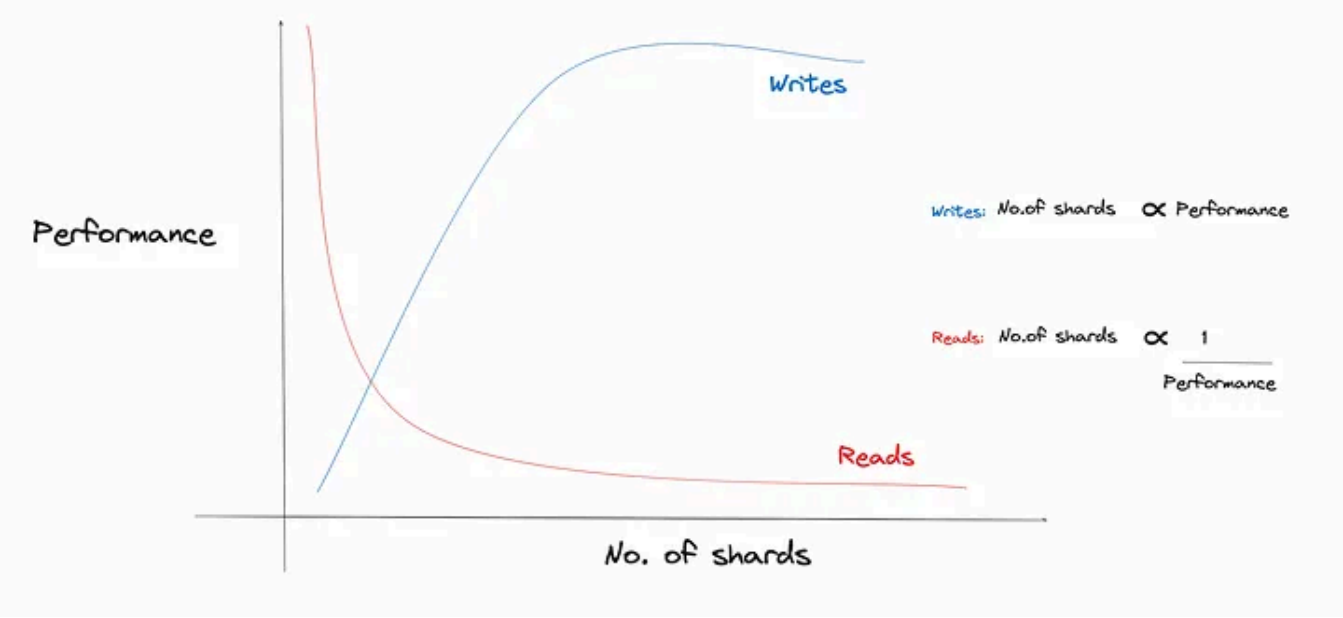

Fig 3.0: No. of shards vs performance for writes and reads

The decision about the number of shards depends on many factors that collectively try to predict the write load on a specific counter in the short term. For tweets, these factors include follower count. The tweet of a user with millions of followers gets more shards than a user with few followers on Twitter because there is a possibility that their tweets will get many, often millions, of likes.

We require that our system can **dynamically expand or shrink the number of shards based on the current need.**

We require that our system can dynamically expand or shrink the number of shards based on the current need. We need to **monitor the write load** for all the shards to appropriately route requests to specific shards, possibly using load balancers. Such a feedback mechanism can also help us decide when to close down some of the shards for a counter and when to add additional shards. This process does not only

provide good performance for the end user but also utilizes our resources at near-optimal levels.

Burst of writes requests

How does the system select these shards operating on different computational units (nodes) to assign the write requests?

Round-robin selection

One way to solve the above problem is to use a round-robin selection of shards. For example, let's assume the number of shards is 100. The system starts with shard_1, then shard_2, and continues until it reaches shard_100. Usually, round-robin work allocation either o**verloads or underutilizes resources** because scheduling is done **without considering the current load conditions**.

Random selection

Another simple approach can be to uniformly and randomly select a shard for writing. The challenge with both round-robin selection and random selection is with variable load changes on the nodes (where shards are hosted). It is hard to appropriately distribute the load on available shards. Load variability on nodes is common because a physical node is often being used for multiple purposes.

Metrics-based selection — The third approach is shard selection based on specific metrics.

Manage read requests

When the user sends the read request, the system will aggregate the value of all shards of the specified counter to return the total count of the feature (such as like or reply). **Accumulating values** from all the shards on each read request **will result in low read throughput and high read latency**.

The decision of when the system will sum all shards values is also very critical. If there is high write traffic along with reads, it might be virtually **impossible** to get a real current value because by the time we report a read value to the client, it will have already changed. So, **periodically reading all the shards of a counter** and **caching** it should serve most of the use cases. By reducing the accumulation period, we can increase the accuracy of read values.

Placement of sharded counters

An important concern is where we should place shared counters. Should they reside on the same nodes as application servers, in separate nodes in the data center, in nodes of CDN at the edge of a network near the end users?

The exact answer to this question depends on our specific use case. For Twitter, we can compute counts by placing sharded counters near the user, which can also help to handle heavy hitter and Top K problems efficiently.

- **Cassandra** — Reads can store counter values in appropriate data stores and rely on the respective data stores for read scalability. The **Cassandra** store can be used to maintain views, likes, comments, and many more counts of the users in the specified region. These counts represent the last computed sum of all shards of a particular counter.
- **Redis or Memcache —** We also need storage for the sharded counters, which store all information about them with their metadata. The **Redis or Memcache** servers can play a vital role here. For example, each tweet's unique ID can become the key, and the value of this key can be a counter ID, or a list of counters' IDs (like counter, reply counter, and so on). Furthermore, each counter ID has its own key-value store where the counter (for example, a likes counter) ID is a key and the value is a list of assigned shards.

Evaluation

Availability

A single counter for any feature (such as like, view, or reply) has a high risk of a single point of failure. Sharded counters eliminate a single point of failure by running many shards for a particular counter. The system remains available even if some shards go down or suffer a fault. This way, sharded counters provide high availability.

Scalability

Sharded counters allow high horizontal scaling as needed. Shards running on additional nodes can be easily added to the system to scale up our operation capacity. Eventually, these additional shards also increase the system's performance.

Reliability

Another primary purpose of the sharded counters is to reduce the massive write request by mapping each write request to a particular shard. Each write request is handled when it comes, and there is no request waiting in the queue. Due to this,

the hit ratio increases, and the system's reliability also increases. Furthermore, the system periodically saves the computed counts in stable storage — Cassandra, in this case.

Conclusions

Nowadays, sharded counters are a key player in improving the overall performance of giant services. They provide high scalability, availability, and reliability. Sharded counters solved significant issues, including the heavy hitters and Top K problems, that are very common in large-scale applications.

Sharded Counters | Software Architect | Software Architecture | Distributed System Design | System Design Interview

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium